Software Watermarking Resilient to Debugging Attacks

Gaurav Gupta, Josef Pieprzyk
Centre for Advanced Computing - Algorithms and Cryptography,
Department of Computing, Division of Information and Communication Sciences,
Macquarie University, Sydney, NSW - 2109,
Australia
Email: {ggupta,josef}@ics.mq.edu.au

***Abstract*— In 2006, Gaurav Gupta and Josef Pieprzyk presented an attack on the branch-based software watermarking scheme proposed by Ginger Myles and Hongxia Jin in 2005. The software watermarking model is based on replacing *jump instructions* or *unconditional branch statements* (UBS) by calls to a *fingerprint branch function* (FBF) that computes the correct target address of the UBS as a function of the generated fingerprint and integrity check. If the program is tampered with, the fingerprint and/or integrity checks change and the target address is not computed correctly. Gupta and Pieprzyk's attack uses debugger capabilities such as register and address lookup and breakpoints to minimize the requirement to manually inspect the software. Using these resources, the FBF and calls to the same is identified, correct displacement values are generated and calls to FBF are replaced by the original UBS transferring controlof the attack to the correct target instruction. In this paper, we propose a watermarking model that provides security against such debugging attacks. Two primary measures taken are shifting the stack pointer modification operation from the FBF to the individual UBSs, and coding the stack pointer modification in the same language as that of the rest of the code rather than assembly language to avoid conspicuous contents. The *manual* component complexity increases from $\mathcal{O}(1)$ in the previous scheme to $\mathcal{O}(n)$ in our proposed scheme.**

***Index Terms*— watermarking, fingerprint, software**

I. Introduction and Related Work

Software piracy and forgery have prompted researchers to investigate watermarking and fingerprinting. The watermark establishes author identity and the fingerprint extracts the buyer's information. Many watermarking and fingerprinting schemes have been proposed with these objectives, and can be classified as follows:

- *Graph-based software watermarking*: Software can be realized as a graph G where nodes of the graph are represented by a sequential instruction set and edges are the branch instructions that transfer control from one instruction to another instruction. Represent a node by $n_i = \{s_{i_1}, \ldots, s_{i_{k_i}}\}$ where k_i is the number of instructions in node n_i. An edge $e_{i,j}$ represents a branch instruction from n_i to n_j.
  Watermark is contained in another independent program, also represented by a graph G'. The purpose of watermark embedder is to add G and G' such that it is computationally infeasible for the attacker to find the right cut that separates G, G'. The watermarked software graph $G_w = G + G'$.
  The first graph-based software watermarking scheme was proposed by Venkatesan et al. [1]. Software and watermark codes are converted to digraphs and extra edges are added between these two digraphs by adding function calls. The authors propose to implement a random walk for the selection of the next node to be visited during the embedding process. But this *walk* is not truly random. Let the next node to be visited be n, nodes remaining in the software be N_s and nodes remaining in the watermark be N_w. $P(n \in G') = \frac{N_w}{N_w+N_s}$ and $P(n \in G) = \frac{N_s}{N_w+N_s}$. Assuming $N_s \gg N_w$ in the beginning, there is a high probability that the nodes selected towards the beginning belong to the software and the nodes selected towards the end belong to the watermark. Hence the watermark is skewed towards the tail of the watermarked program. Other papers in graph-based software watermarking are available for readers' consideration in [2]–[6]. Instruction and block re-ordering attacks remain to be a problem for all these models.
- *Register-based software watermarking*: Watermark is embedding in the order of the registers that are used to store variables. Register-based software watermarking based on the QP algorithm (named after authors Qu and Potkonjak) [7], [8] is presented in [9]. If two variables are required at the same time, it does not matter which register stores which one of the two variables. Thus the registers that store the two variables can be swapped. The watermark is encoded in the ordering of registers. Register reallocation is an obvious and direct attack against such watermarking scheme. Secondary watermarking also destroys the old watermark.
- *Thread-based software watermarking*: Just like they are encoded in the ordering of registers in the previous case, watermarks are encoded in the ordering of threads in this case [10]. If there are 2 threads; Ta, T_b, $T_a \rightarrow T_b$ encodes watermark $(00)_2$, $T_b \rightarrow T_a$ encodes $(01)_2$, $T_a \rightarrow T_a$ encodes $(10)_2$ and $T_b \rightarrow T_b$ encodes $(11)_2$. The threads that execute program

code can be manipulated by the attacker in a bid to destroy the watermark.

- *Obfuscation-based software watermarking*: Object-oriented programs can be watermarked using this watermarking model [11]–[13]. Class C that performs n functions $\{f_1, \ldots, f_n\}$ is divided into m subclasses $\{C_1, C_2, \ldots, C_m\}$. The watermark is encoded in which subclasses contain which functions.
- *Branch-based software watermarking*: Unconditional branch statements (UBS) are converted to function calls to a special function called the branch function. The purpose of branch function is to transfer the control to the correct target address of the UBS [2]. A branch is represented as $l_a \rightarrow l_b$ indicating a jump from instruction l_a to instruction l_b. If the program contains $l_{begin} \rightarrow l_{end}$, the control-flow graph is changed to $l_{begin} \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow l_{end}$ by adding multiple intermediate instructions (l_{begin} jumps to a_1, a_1 that jumps to a_2 and so on). The following condition is checked while inserting these instructions.
  If watermark bit $w_i = 1$**,**
  then $address(a_{i+1}) > address(a_i)$**,**
  otherwise $address(a_{i+1}) < address(a_i)$
  Secondly the jump instructions are replaced by branch function calls. The branch function determines the correct target address based on the calling address.
  If an attacker deletes one or more intermediate instructions and/or adds his/her own bogus intermediate instructions, the execution path is modified thereby modifying the watermark bit-string that is recovered.
  To work around such problems and others, Myles and Jin propose a fingerprinting scheme in [14]. The concept of a *branch function* transferring control to the target of the UBS remains common to their model, but their branch function generates the fingerprint, and is therefore called a *Fingerprint Branch Function* (FBF). To verify that the attacker hasn't modified the rest of the code, FBF can also check for integrity of the code. Details of this scheme are discussed the Section III.

II. Requirements

This section describes requirements of a watermarking scheme, general attacks on watermarked media and other attacks specifically against watermarked software. Any software watermarking model should consider security of the scheme against these attacks individually, and in combination.

The requirements of a watermarking scheme are as follows,

1) ***Imperceptibility:*** Watermark insertion should cause minimum time and space overload on the software. The scheme should also avoid watermark's explicit storage inside the software.
2) ***Watermark detectability:*** The watermark should be able to be detected with a probability $1 - \epsilon$ where, ideally speaking, $\epsilon \approx 0$. Probability of false positives should also be negligible.
3) ***Robustness:*** Watermark should survive generic attacks and those specific to software watermarks. These attacks as listed below,
   a) **Additive Attack:** Insertion of variables and instructions to the watermarked software to disturb the watermark encoding.
   b) **Subtractive Attack:** Removal of (non-essential) variables and code to probabilistically delete the watermark.
   c) **Secondary Watermarking Attack:** Insertion of attacker's own watermark in a bid to overwrite the original watermark. Security against this attack is typically provided by the secret key which generates location of watermark.
   d) **Decompilation-Recompilation Attack:** This attack is specific to software and recompiles the software after decompiling it. Numerous compilation-specific orderings are modified in this manner.
   e) **Invertibility Attack:** Attempting to detect accidental presence of a watermark in the software by trying random input sequences.
   f) **Code re-writing Attack:** Parts of the code may be re-written by the attacker to minimize possibility of watermark being encoded in language syntax.
   g) **Register renaming Attack:** Renaming the registers to destroy watermark encoded in register ordering [7], [8].

III. Myles and Jun's watermarking scheme

Branch statements are replaced by calls to an FBF which returns control to the target address. Each time the FBF is called, it calculates a secret key used to generate the target address. This key is calculated using a hash function of the old key (key derived in the previous stage), the authorship mark, and the integrity check on some section of the code (a different section during each stage). An integrity check branch function (ICBF) is appended to the program to ensure that the attacker doesn't modify the FBF. When a watermarked program is manipulated, the generated keys and/or integrity check values change and hence the target address is incorrectly calculated. The program behaves in manner dependent on the resulting target address - if the target address belongs to the code section, program continues to execute but incorrectly and if the target address lies outside the code section (say the data section), the program returns with a run-time error. The watermarking model can be seen as a system comprising of two algorithms - *embed* and *recognize*.

- $P \rightarrow$ original software,
- $AM \rightarrow$ authorship mark,
- $key_{AM} \rightarrow$ secret input sequence,
- $key_{FP} \rightarrow$ initial secret key,

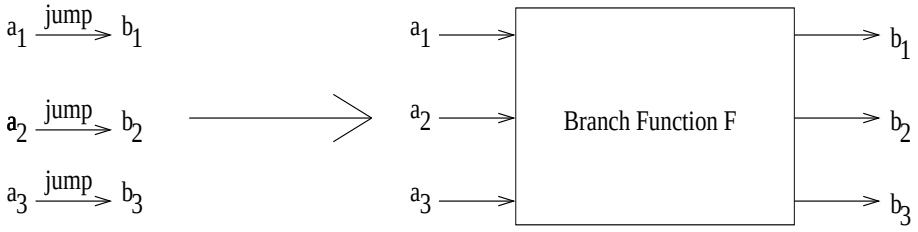

Figure 1. Fingerprint branch function changes the return address based on calling address and transfers control to the target instruction of the original unconditional branch statement. Unconditional branch statements are then replaced by calls to $\mathcal{FBF}$.

- $FP \rightarrow$ fingerprint,
- $P_w \rightarrow$ watermarked software

1) embed$(P, AM, key_{AM}, key_{FP}) \rightarrow P_w, FP$
2) recognize$(P_w, key_{AM}, key_{FP}) \rightarrow AM, FP$

The embed algorithm is explained below,

1) Let F be the set of functions that lie in the execution path when the program is run with input sequence key_{AM}, let $\bar{F}$ be the set of the remaining functions in P.
2) Let the number of unconditional branch statements in functions in F and $\bar{F}$ be n and m respectively.
3) Add two arrays to the data section of the program - DT_F of size n and $DT_{\bar{F}}$ of size m.
4) Let the displacement between the source and target of the i^{th} unconditional branch statements in functions from F be $d_i = t_i - s_i$, where s_i is the source/origin and t_i is the target/destination of the UBS.
5) In P, insert fingerprint branch function $\mathcal{FBF}$ which implements the following steps,
   a) $k_0 = key_{FP}$.
   b) For $1 \leq i \leq n$,
      i) Check integrity value $\mathcal{IC}_i$ of code section $\mathcal{S}_i$.
      ii) Generate k_i using $k_{i-1}, \mathcal{IC}_i, AM$ from the one-way hash function SHA_1

$$k_i = SHA_1[(k_{i-1} \bigoplus AM) \| \mathcal{IC}_i] \quad (1)$$

   c) $DT_F[h_1(k_i)] = d_i$, where h_1 is a hash function mapping the keys to the indices, $h_1 : \{k_1, k_2, \ldots, k_n\} \rightarrow \{1, 2, \ldots, \tilde{n}\} (n \leq \tilde{n})$.
6) Let the displacement between the source and target of i^{th} unconditional branch statements in functions from $\bar{F}$ be $\bar{d}_i = \bar{t}_i - \bar{s}_i$, where $\bar{s}_i$ is the source/origin and $\bar{t}_i$ is the target/destination of the UBS.
7) Insert integrity check branch function $\mathcal{ICBF}$ in P that performs following steps,
   a) Compute integrity check value $\mathcal{IC'}_i$ for code containing $\mathcal{FBF}$.
   b) $DT_{\bar{F}}[h_2(\mathcal{IC'}_i)] = \bar{d}_i$, where h_2 is a hash function mapping the integrity checks to the indices, $h_2 : \{\mathcal{IC'}_1, \mathcal{IC'}_2, \ldots, \mathcal{IC'}_m\} \rightarrow \{1, 2, \ldots, \tilde{m}\} (m \leq \tilde{m})$.
8) Replace unconditional branch statements in F and $\bar{F}$ by calls to $\mathcal{FBF}$ and $\mathcal{ICBF}$ respectively.

The fingerprint is a concatenation of generated keys.

$$FP = k_1 \| k_2 \| \ldots \| k_n \quad (2)$$

Since each user has a distinct initial key $key_F M$, each user also has a distinct fingerprint. Thus the embed algorithm satisfies the basic condition of a fingerprint that no two users should have the same fingerprint.

The watermarked program P_w, secret input key_{AM} and the initializing key key_{FP} are provided to the *recognize* algorithm which returns the authorship mark AM and fingerprint FP. Running P_w with input sequence key_{AM} executes functions in F generating the fingerprint $FP = k_1 \| k_2 \| \ldots k_n$ (initializing $k_0 = key_{FP}$) using Equation 4. AM is recovered from the one-way hash function $k_i = SHA_1[(k_{i-1} \bigoplus AM) \| \mathcal{IC}_i]$.

IV. Gupta and Pieprzyk's Debugging Attack

In 2006, Gupta and Pieprzyk presented an attack on Myles and Jun's watermarking scheme [15]. Major portion of this attack is automated and manual inspection is kept to a minimal making it an extremely practical attack. In this section, this attack is described in detail.

The attacker's goal is to restore the original program P from the fingerprinted program P_w. The transfer of control from source to target of an unconditional branch statement is through $\mathcal{FBF}$. Inside $\mathcal{FBF}$, the displacement added to/subtracted from the source address is calculated as a hash of the generated key. This key itself is calculated using the previous stage's key, authorship mark and an integrity check value. If any one of these values are wrong, the new key, and thereby the displacement is incorrect which results in $\mathcal{FBF}$ returning control to the wrong instruction. $\mathcal{ICBF}$ verifies the integrity of $\mathcal{FBF}$, providing a second layer of security.

Gupta and Pieprzyk's attack targets the dependence of target address on key generated. If the execution path can be made independent of the keys generated, $\mathcal{FBF}$ and $\mathcal{ICBF}$ can be deleted. Gupta and Pieprzyk propose to track register values, including the stack pointer (SP) at:

1) Entry point of $\mathcal{FBF}$: SP = sp_{i_1}
2) Exit/ Return instruction of $\mathcal{FBF}$: SP = sp_{i_2}

Displacement value d_i is given by the difference $sp_{i_2} - sp_{i_1}$. Identifying instructions participating in fingerprint generation is also achievable. The attacker can create a mapping of functions being called by other functions and thereby create sets of functions which all point to

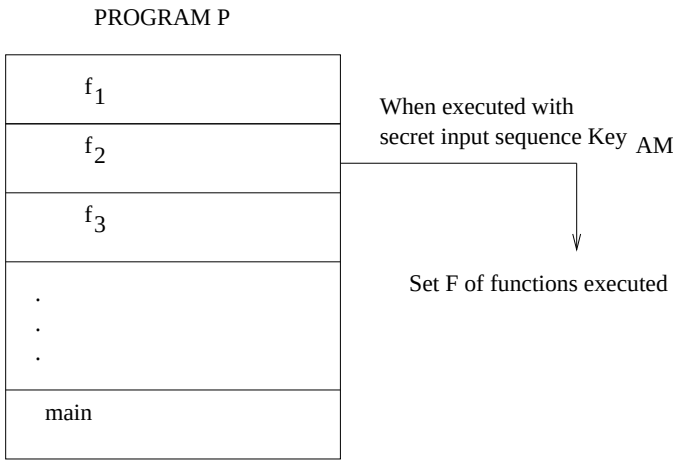

Figure 2. The set F of functions is executed when the program P is executed with the input key_{AM}. Effectively, key_{AM} generates a trace of the program P.

one particular function. $\mathcal{FBF}$ is identifiable by short-listing stack-pointer modifying functions and presence of assembly level code in the program. The set F is the set of functions that call $\mathcal{FBF}$. Since key_{AM} is not needed to identify the set of functions participating in watermarking, Gupta and Pieprzyk eliminate the requirement of key_{AM}. In functions belonging to F, an instruction calling $\mathcal{FBF}$ is now replaced by an unconditional branch to the instruction to which $\mathcal{FBF}$ would return control.

After changing $\mathcal{FBF}$ and $\mathcal{ICBF}$ calls by unconditional branch statements to the correct target instruction, the two functions ($\mathcal{FBF}, \mathcal{ICBF}$) are deleted. When the *recognize* algorithm is run with input key_{AM}, key_{FP}, the inputs are unused dead variables, the algorithm doesn't output the fingerprint FP and the recognition algorithm fails. The resulting software is equivalent to an un-watermarked software.

The attacker may be interested in superimposing his/her own watermark after deleting the original watermark. To achieve this purpose, (s)he can insert a new fingerprint branch function that outputs the attacker's authorship mark when the program is executed with key_AM.

Summarizing the process, the steps performed by the attacker are:

1) *Locate* $\mathcal{FBF}$: This is achieved through the following characteristics of $\mathcal{FBF}$,
   a) Presence of assembly level code (For example *_asm* blocks in C and C++.
   b) Mismatched values of Stack Pointer at entry and exit of the function.
2) *Locate* F: Only functions in F call $\mathcal{FBF}$ and hence locating them is a trivial task once $\mathcal{FBF}$ is located.
3) *Computing displacements*: Values of stack pointer at the first and the last statement in $\mathcal{FBF}$; sp_{i_1} and sp_{i_2} respectively; is noted. Displacement d_i is the difference of these two values ($sp_{i_2} - sp_{i_1}$).
4) *Replacing* $\mathcal{FBF}$ *calls to unconditional branch statements*: The correct target instruction can be located using the source instruction and the displacement.
5) *Deleting* $\mathcal{FBF}$ *and* $\mathcal{ICBF}$: Locating $\mathcal{ICBF}$ is similar to locating $\mathcal{FBF}$. Then, both these functions are deleted.
6) *Creating a modified watermarked program*: The attacker may want to embed his/her own authorship mark $\tilde{AM}$. For this to be effective, *recognize*$(\tilde{P}, key_{FP}, key_{AM})$ should return $\tilde{AM}, \tilde{FP}$) as output where $\tilde{P}$ is the attacked and modified program and $\tilde{FP} \neq FP$. Following steps illustrate the process of secondary watermarking.
   a) For all unconditional branch statements for all functions in F, store the displacements (between the calling address and the target address) in an two-dimensional array D along with the calling address.
   b) Replace the unconditional branch statements by call to a new fingerprint branch function, $\tilde{\mathcal{FBF}}$.
   c) $\tilde{\mathcal{FBF}}$ **need not** generates a key based on the previous key and attacker's authorship mark $\tilde{AM}$.

$$\tilde{k}_i = SHA1[\tilde{k}_{i-1} \bigoplus \tilde{AM}]. \qquad (3)$$

   Note that the generation of new keys no longer requires the attacker to compute integrity check.
   d) A hash function h maps the keys to the correct displacements, $h : \{\tilde{k}_1, \tilde{k}_2, \ldots, \tilde{k}_n\} \rightarrow \{1, 2, \ldots, m\} (n \leq m)$. The table $\tilde{T}$ storing the correct displacements is inserted in the data section of the program.
   $\tilde{T}[h(\tilde{k}_i)] = d_i$

Fingerprint $\tilde{FP}$ generated is different from the original key sequence FP as the individual keys are different.

Proof: $\tilde{k}_i \neq k_i, 1 \leq i \leq n$
$\Rightarrow \{\tilde{k}_1, \tilde{k}_2, \ldots, \tilde{k}_n\} \neq \{k_1, k_2, \ldots, k_n\}$
$\Rightarrow \{\tilde{k}_1, \tilde{k}_2, \ldots, \tilde{k}_n\} \neq FP$

$\Rightarrow \tilde{FP} \neq FP$ ■

Algorithm *recognize*($\tilde{P}, key_{FP}, key_{AM}$) will output $\tilde{FP}, \tilde{AM}$ upon execution establishing the attacker's ownership over the software.

V. Surviving the Debugging Attack

In this section, we analyze the debugging attack discussed in the previous section and identify the assumptions on which the attack is based. Further, we shall propose way(s) to counter-attack these assumptions and thus secure the software from any attempts of removing or modifying the watermark.

The basic assumption we take is that the source code is available to the attacker for inspection. This is a strong assumption taking into account that most commercial softwares do not come with the source codes. However, we take into consideration the growing popularity of open source software as well. Plus, having a stronger assumption and thereby an easier attack, our watermarking scheme results in getting stronger (if it can survive the attacks). Manual inspection of the source code is practically infeasible, given that it can run into hundreds of thousands of code lines. Thus the attacker tries to minimize the size of source code that (s)he manually inspects. Debugging mechanisms provide strength to the attacks in such cases by reducing the size of code to be inspected to potentially a few hundred lines. According to Gupta and Pieprzyk [15], in order to identify $\mathcal{FBF}$, the attacker relies on either,

1) $\mathcal{FBF}$ containing assembly level code,
2) Stack Pointer value differing at entry and exit of $\mathcal{FBF}$.

We have a typical scenario where a source instruction I_s needs to transfer control to a target instruction I_t. I_s calls $\mathcal{FBF}$ which manipulates the stack pointer and returns the control to I_t. Addressing the first indicator, the code that performs stack pointer modifications can always be written in a higher language and thus it is not necessary that $\mathcal{FBF}$ contains assembly level code.

The stack pointer modification in $\mathcal{FBF}$ is the basis of Gupta and Pieprzyk's attack. If $\mathcal{FBF}$ does not perform this extremely visible and conspicuous stack pointer modification and only returns the value of generated key to I_s, then I_s can add compute the displacement, add it to the stack pointer and transfer control to I_t. Given this process, the attack $\mathcal{FBF}$ cannot be identified and all subsequent steps of the attack fail. Another advantage is that the attacker can no longer get the values of all displacement values by placing two breakpoints at the start and end of $\mathcal{FBF}$ (which is the case in [14]). We can achieve this task by shifting the stack pointer modification instruction from $\mathcal{FBF}$ to function containing I_s.

To get the values of displacements in the new model, the attacker would need to place breakpoints before and after each source instruction I_s. In the previous model, the attacker would have had to place only two breakpoints; at the start and end of $\mathcal{FBF}$ irrespective of the number of unconditional branch statements, and thus the amount of work attacker had to do was independent of code size. But now the amount of work attacker needs to perform manually is directly proportional to the number of unconditional branch statements present in functions from F.

Another strong assumption in Gupta and Pieprzyk's attack is that only unconditional branch statements in functions belonging to F call $\mathcal{FBF}$. If certain bogus calls to $\mathcal{FBF}$ are inserted while embedding the watermark, this assumption in not true. Thus identifying F is not possible for the attacker through methods pointed out by Gupta and Pieprzyk. The action taken by $\mathcal{FBF}$ when called by these bogus statements is pre-defined during embedding.

The modified algorithm $embed_2$ is given below,

1) Let F be the set of functions that lie in the execution path when the program is run with input sequence key_{AM}, let $\bar{F}$ be the set of the remaining functions in P.
2) Let the number of unconditional branch statements in functions in F and $\bar{F}$ be n and m respectively.
3) Add two arrays to the data section of the program - DT_F of size n and $DT_{\bar{F}}$ of size m.
4) Let the displacement between the source and target of the i^{th} unconditional branch statements in functions from F be $d_i = t_i - s_i$, where s_i is the source/origin and t_i is the target/destination.
5) In P, insert fingerprint branch function $\mathcal{FBF}$ which implements the following steps,
   a) $k_0 = key_{FP}$.
   b) For $1 \leq i \leq n$,
      i) Check integrity value $\mathcal{IC}_i$ of section $\mathcal{S}_i$ of code.
      ii) Generate k_i using $k_{i-1}, \mathcal{IC}_i, AM$ from the one-way hash function SHA_1.

$$k_i = SHA_1[(k_{i-1} \bigoplus AM) \| \mathcal{IC}_i] \quad (4)$$

6) Return k_i to the calling instruction I_s.
7) I_s looks up the displacement that is indexed by hash of the key and stored in table DT_F in data section of the program. $DT_F[h_1(k_i)] = d_i$, where h_1 is a hash function mapping the keys to the indices, $h_1 : \{k_1, k_2, \ldots, k_n\} \rightarrow \{1, 2, \ldots, \tilde{n}\} (n \leq \tilde{n})$.
8) I_s transfers control to $sp+d_i$ where sp is the current value of stack pointer, using higher language code.
9) Insert integrity check branch function $\mathcal{ICBF}$ in P that checks integrity $\mathcal{IC}'_i$ of code containing $\mathcal{FBF}$
10) Return $\mathcal{IC}'_i$ to the calling instruction $\bar{I}_s$.
11) $\bar{I}_s$ looks up the displacement that is indexed by hash of the key and stored in table $DT_{\bar{F}}$ in data section of the program. $DT_{\bar{F}}[h_2(\mathcal{IC}'_i)] = \bar{d}_i$, where h_2 is a hash function mapping the integrity checks to the indices, $h_2 : \{\mathcal{IC}'_1, \mathcal{IC}'_2, \ldots, \mathcal{IC}'_m\} \rightarrow \{1, 2, \ldots, \tilde{m}\} (m \leq \tilde{m})$.
12) $\bar{I}_s$ transfers control to $sp+d_i$ where sp is the current value of stack pointer, using higher language code.

13) Replace unconditional branch statements in F and $\bar{F}$ by calls to $\mathcal{FBF}$ and $\mathcal{ICBF}$ respectively.

Steps 6–8 and 10–12 are modified such that the control transfer is shifted from $\mathcal{FBF}$ and $\mathcal{ICBF}$ to F and $\bar{F}$ respectively.

VI. Analysis

There are two primary modifications to the watermarking algorithm of Myles and Jun,

1) Strict usage of higher language code to perform stack pointer modifications.
2) Shifting the stack pointer modification from the fingerprint branch function $\mathcal{FBF}$ to the source instruction of the unconditional branch statement.

Figure 3 and 4 illustrate the differences between the watermarking scheme of Myles and Jun and our proposed scheme. While in the former, $\mathcal{FBF}$ performs the key, displacement and target address computation, in the latter scheme it only performs key computation and returns the value of the key to the source instruction. The source instruction then computes target address as a function of displacement, which in turn is computed from the key. Thus an attacker has to place n pairs of breakpoints to find the correct target addresses. Thus, *manual* component complexity increases from $\mathcal{O}(1)$ in the previous scheme to $\mathcal{O}(n)$ in our proposed scheme. Security against other attacks such as additive or subtractive attacks remains the same as in [14].

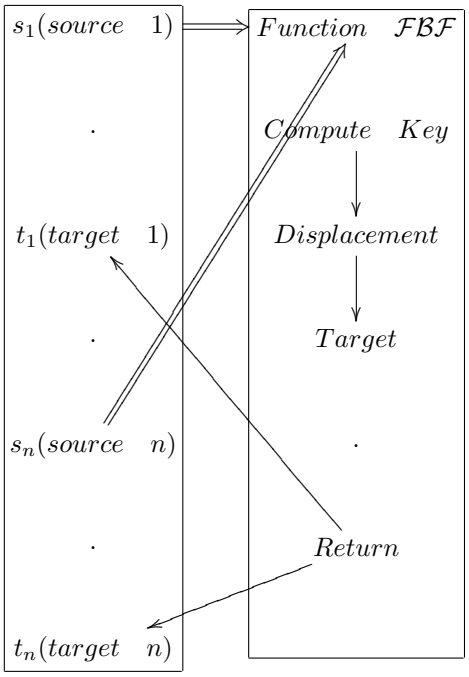

Figure 3. In the previous scheme by Myles and Jun, source instruction calls the fingerprint branch function which computes the key, displacement and target address, in that order, and returns the control to the target instruction

VII. Conclusion

In this paper, we have presented modifications on [14] so that the watermarking scheme can withstand debugging attacks like the one suggested in [15]. The attacker needs much more extensive manual inspection ($\emptyset(n)$) of the watermarked program in order to remove the watermark. This can prove to be infeasible given that software sizes can easily run into thousands of lines of code. The key to surviving the attack is shifting the stack pointer manipulation operation from the fingerprint branch function to the original unconditional branch statements.

The proposed watermarking scheme makes it more difficult and more tedious for an attacker to locate manipulative functions such as $\mathcal{FBF}, \mathcal{ICBF}$ but does not rule out eventual location and deletion of these functions. It is desirable to formulate a watermarking scheme belonging to the family of stack modifying functions that is completely secure against debugging attacks. To accomplish this task, one needs to hide the dependency of target instruction calculation on key generation process from the user. This is an open problem in the field of branch based software watermarking.

References

[1] R. Venkatesan, V. Vazirani, and S. Sinha, "A graph theoretic approach to software watermarking," in *Proceedings of 4th Information Hiding Workshop, LNCS*, vol. 2137, 2001, pp. 157–168.

[2] C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn, and M. Stepp, "Dynamic path-based software watermarking," in *Proceedings of Conference on Programming Language Design and Implementation*, vol. 39, June 2004, pp. 107–118.

[3] C. Collberg, A. Huntwork, E. Carter, and G. Townsend, "Graph theoretic software watermarks: Implementation, analysis, and attacks," in *Proceedings of 6th Information Hiding Workshop, LNCS*, vol. 3200, 2004, pp. 192–207.

[4] C. Collberg, S. Kobourov, E. Carter, and C. Thomborson, "Error-correcting graphs for software watermarking," in *Proceedings of 29th Workshop on Graph Theoretic Concepts in Computer Science*, 2003, pp. 156–167.

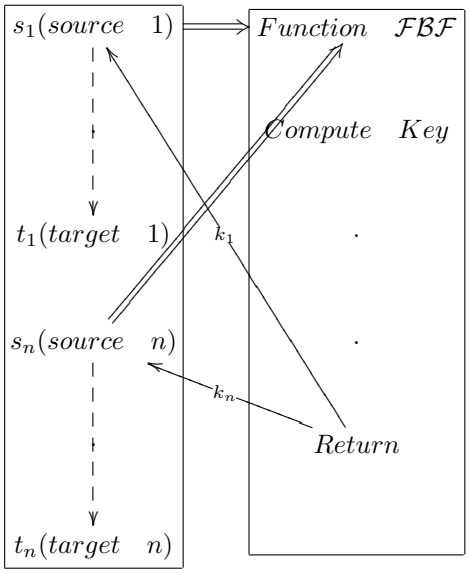

Figure 4. In our proposed scheme, source instruction calls the fingerprint branch function which computes the key and returns the key back to the source instruction. The source instruction then computes the target address using this key and transfers control to the target instruction.

[5] C. Collberg and C. Thomborson, "Software watermarking: Models and dynamic embeddings," in *Proceedings of Principles of Programming Languages 1999, POPL'99*, 1999, pp. 311–324. [Online]. Available: citeseer.ist.psu.edu/collberg99software.html
[6] C. Thomborson, J. Nagra, R. Somaraju, and C. He, "Tamper-proofing software watermarks," in *Proceedings of Australasian Information Security Workshop*, vol. 32, 2004, pp. 27–36.
[7] G. Qu and M. Potkonjak, "Analysis of watermarking techniques for graph coloring problem," in *Proceedings of International Conference on Computer Aided Design*, 1998, pp. 190–193. [Online]. Available: citeseer.ist.psu.edu/qu98analysis.html
[8] ——, "Hiding signatures in graph coloring solutions," in *Proceedings of 3rd Information Hiding Workshop, LNCS*, vol. 1768, 1999, pp. 348–367.
[9] G. Myles and C. Collberg, "Software watermarking through register allocation: Implementation, analysis, and attacks," in *Proceedings of International Conference on Information Security and Cryptology, LNCS*, vol. 2971, 2003, pp. 274–293.
[10] J. Nagra and C. Thomborson, "Threading software watermarks," in *Proceedings of 6th Information Hiding Workshop, LNCS*, vol. 3200, 2004, pp. 208–223.
[11] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation - tools for software protection," in *IEEE Transactions on Software Engineering*, vol. 28, no. 8, August 2002, pp. 735–746. [Online]. Available: citeseer.ist.psu.edu/collberg02watermarking.html
[12] K. Fukushima and K. Sakurai, "A software fingerprinting scheme for java using classfiles obfuscation," in *Proceedings of Information Security Applications, LNCS*, vol. 2908, 2004, pp. 303–316.
[13] M. Sosonkin, G. Naumovich, and N. Memon, "Obfuscation of design intent in object-oriented applications," in *Proceedings of 3rd ACM workshop on Digital Rights Management*, 2003, pp. 142–153.
[14] G. Myles and H. Jin, "Self-validating branch-based software watermarking." in *Proceedings of 7th Information Hiding Workshop, LNCS*, vol. 3727, 2005, pp. 342–356.
[15] G. Gupta and J. Pieprzyk, "A low-cost attack on branch-based software watermarking schemes." in *IWDW*, 2006, pp. 282–293.

Gaurav Gupta is currently a Ph.D. candidate at Macquarie University, NSW, Australia. Gaurav Gupta received his Bachelor of Computer Applications from Devi Ahilya University, Indore, India in 2002 and Master of Computing degree from National University in Singapore in 2004.

He has teaching experience in programming languages, Operating Systems and Information Systems. His research interests include multimedia copyright protection, digital watermarking and information security.

Josef Pieprzyk received B.Sc in Electrical Engineering from Academy of Technology in Bydgoszcz, Poland, M.Sc in Mathematics from University of Torun, Poland, and PhD degree from Polish Academy of Sciences in Warsaw. Josef Pieprzyk is a Professor in the Department of Computing, Macquarie University, Sydney, Australia.

He has published 5 books, edited 10 books (conference proceedings published by Springer-Verlag), 3 book chapters, and around 160 papers in refereed journals and refereed international conferences. He is a member of the editorial board for International Journal of Information Security (Springer-Verlag), Journal of Mathematical Cryptology (W de Gruyter), International Journal of Security and Networks, and International Journal of Information and Computer Security. He has served as Program Chair for 8 international conferences and member of Program Committees for more than 30 international conferences. His research interest includes computer network security, database security, design and analysis of cryptographic algorithms, algebraic analysis of block and stream ciphers, theory of cryptographic protocols, secret sharing schemes, threshold cryptography, copyright protection, e-Commerce and Web security.

Dr. Pieprzyk is a member of IACR.